CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 0000; **00**:1–20
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe

Nonblocking Collectives for Scalable Java Communications

Sabela Ramos[1]*, Guillermo L. Taboada[1], Roberto R. Expósito[1] and Juan Touriño[1]

[1] *Computer Architecture Group, Department of Electronics and Systems, University of A Coruña, Spain*

SUMMARY

This paper presents a Java implementation of the recently published MPI 3.0 nonblocking message passing collectives in order to analyze and assess the feasibility of taking advantage of these operations in shared memory systems using Java. Nonblocking collectives aim to exploit the overlapping between computation and communication for collective operations to increase scalability of message passing codes, as it has been done for nonblocking point-to-point primitives. This scalability has become crucial not only for clusters but also for shared memory systems due to the current trend of increasing the number of cores per chip, which is leading to the generalization of multi- and many-core processors. Message passing libraries based on RDMA (Remote Direct Memory Access), thread-based progression or implementing pure multi-threading shared memory support could potentially benefit from the lack of imposed synchronization by nonblocking collectives. But, although the distributed memory scenario has been well studied, the shared memory one has not been tackled yet. Hence, nonblocking collectives support has been included in FastMPJ, a Message Passing in Java (MPJ) implementation, and evaluated on a representative shared memory system, obtaining significant improvements due to overlapping and lack of implicit synchronization, and with barely any overhead imposed over common blocking operations.

Received . . .

KEY WORDS: Collective Operations; Java Multi-threading; Message Passing in Java (MPJ); MPI 3.0; Nonblocking Collectives; Shared Memory architectures

1. INTRODUCTION

Communication may become one of the major bottlenecks in the scalability of parallel codes, especially when increasing the number of cores involved. Message passing has tried to avoid this overhead by overlapping communication and computation via nonblocking point-to-point primitives. Nevertheless, regarding collective operations, only blocking primitives were supported by the MPI standard, forcing programmers to implement their own collective communications involving nonblocking point-to-point primitives when needed. This imposes higher costs of development, risks of bugs and lack of efficiency as it is not possible to take advantage of the highly optimized collective algorithms included in message passing libraries, that usually exploit the underlying hardware. Thus, nonblocking collectives had been proposed to be part of the MPI 2.2 standard but they were postponed until MPI 3.0. To support their inclusion, there is an implementation of these primitives which is compatible with Open MPI called LibNBC [1] that has shown great improvements in collective performance for InfiniBand with a large number of processes.

*Correspondence to: Facultade de Informática, Campus de Elviña s/n, 15071 A Coruña, Spain. E-mail:{sramos,taboada,rreye,juan}@udc.es

Java is the main language in industry and academia and it is increasingly being adopted by the High Performance Computing (HPC) community due to its appealing features such as multi-thread and networking support in the core of the language, and its improvements in performance, which makes it competitive regarding natively compiled languages like C/C++. Thus, different high performance Java projects have emerged in the past few years [2], including several implementations of message passing libraries like FastMPJ [3, 4], which provides high performance communications for different devices, from InfiniBand to shared memory transfers. Since its first release, FastMPJ has paid special attention to collective operations due to their wide use in parallel codes [5] and has provided the user with a set of optimized multi-core aware algorithms for each operation that can be selected at runtime regarding the number of processes and the message size.

The increase in the number of cores per processor, going towards the generalization of multi- and many-core systems, has addressed the scalability in shared memory environments as a crucial issue for parallel computing. In recent works, the implementation of a specific shared memory collective library for the multi-thread communication device `smdev` [6] has also been explored, and it showed great performance but highlighted the costs of synchronization among threads. Blocking collectives impose an implicit synchronization which can be avoided by the adoption of a nonblocking paradigm for these primitives. Thus, this paper presents a nonblocking collective library for message passing in Java that has been developed and benchmarked to show that both overlapping of communication and computation, and the avoidance of extra synchronization improve performance of message passing codes on shared memory systems.

As it has been mentioned, nonblocking collectives for distributed memory architectures have already been presented [1] in order to support their inclusion in the MPI 3.0 standard. However, in spite of the increasing need for scalable solutions for shared memory architectures, nonblocking collectives have not been analyzed yet in multi- and many-core systems. In this scenario, this paper aims to provide generic support for nonblocking collectives in Message-Passing in Java but paying special attention to assess their feasibility for shared memory architectures.

This paper is organized as follows. Section 2 introduces related work. Section 3 describes the design, implementation and operation of the developed nonblocking collective library for multi-core systems. Section 4 presents the performance analysis of the nonblocking collectives on a Sandy-Bridge shared memory system. Section 5 summarizes our concluding remarks.

2. RELATED WORK

Message passing is a well-known paradigm for parallel programming which is widely used for HPC due to its generally good scalability and performance. With the increase in the number of cores per processor, the optimization of message passing libraries for exploiting multi-core shared memory architectures has become a necessity since they provide higher scalability than traditional shared memory paradigms. In fact, MPI libraries such as Open MPI [7] and MPICH2 [8, 9], and MPJ implementations like MPJ Express [10] and FastMPJ [6], include custom communication devices which exploit shared memory transfers. The advantage of Java over traditional languages in HPC (C, Fortran) is that it supports multi-threading in the core of the language and shared memory programming naturally emerges from it, whereas natively compiled languages have to rely on the shared resources management of the operating systems. Our work takes advantage of Java multi-threading for shared memory systems, being built upon FastMPJ shared memory support.

Moreover, when the number of communicating processes is large, synchronous communications impose a high overhead that can be overcome by overlapping communication and computation using asynchronous communications. The benefits of overlapping in message passing libraries have been well studied: e.g. in [11] the authors analyze the benefits for an MPI library which supports overlap with offloading and independent progress; a benchmark to assess the ability of hardware and software to overlap MPI communication and computation is presented in [12], whereas a theoretical analysis for scientific applications is shown in [13] and the benefits of overlapping in an MPI-2 application are evaluated in [14]. Thus, message passing libraries provide nonblocking point-to-point primitives to support asynchronous communications.

DOI: 10.1002/cpe

Since its inception, MPI aimed to provide asynchronous communications. In fact, the authors of [15], while describing the MPI standard, mention that not only nonblocking point-to-point primitives but also collective ones might be useful and should be included in subsequent versions. Nonblocking collectives were attempted to be part of the MPI 2.2 standard but they were finally put off since it would suppose a major change which would fit best with the recently released MPI 3.0 [16] and thus current MPI projects are including the standard nonblocking collectives. However, these collectives have been explored ever since, and different MPI implementations provided their own suite of nonblocking collective operations. Some examples are Adaptive MPI [17], that extends MPI to use virtual processors; the optimization of MPI collectives for the MPICH2-based library used in BlueGene/L [18] and BlueGene/P [19, 20]; or the Component Collective Messaging Interface (CCMI) [21], which is not an MPI implementation but provides a messaging interface with nonblocking collectives. The potential benefits of nonblocking collectives in different applications are analyzed in [22], and how noise affects MPI performance is studied in [23] concluding that nonblocking collectives can help, and the authors demonstrate this statement empirically through the evaluation of their own nonblocking allreduce implementation. 3D FFT (Fast-Fourier Transform) has been stated to be able to take advantage of nonblocking collectives in [24] and [25]. Moreover, there is also a large number of projects which intend to provide low level nonblocking support. In [26] the authors present the implementation of a nonblocking broadcast that takes advantage of the Mellanox ConnectX-2 InfiniBand adapters that offer a task-offload interface (CORE-Direct), being evaluated with the High Performance Linpack (HPL) benchmark. PAMI, a low level messaging interface is extended in [27] to support the implementation of nonblocking collectives in Power7 IH supercomputers. KACC [28] is a new nonblocking communication facility implemented in the OS kernel interrupt context to perform nonblocking asynchronous collective operations without the help of an extra thread, and it is moved to the user level in uKACC [29], which uses the Marcel thread library and the PIOMan's scheduler from Madeleine [30, 31] to implement nonblocking collectives. In [32] the authors discuss a possible implementation of the flexible Group Operation Assembly Language (GOAL) framework to support nonblocking collectives. Additionally, PGAS languages like Unified Parallel C (UPC) also support them [33].

One of the most relevant projects related to nonblocking collectives is LibNBC [1, 34], a nonblocking collective library which is being integrated in Open MPI. In its first version, each operation needs user interaction to progress, but micro-benchmark results show that overlapping computation and communication in collective operations could potentially provide significant performance improvements. The authors state the benefits that nonblocking collectives could add to MPI and show benchmarking results of their implementation, based on avoiding implicit synchronization and taking advantage of nonblocking features of modern network hardware. This implementation aimed to support a strong case in favor of the inclusion of nonblocking collectives in the MPI standard. The library was optimized for InfiniBand in [35], and the benefits and drawbacks of including an extra thread to manage progression instead of user interaction were evaluated in [36]. This work compares a polling strategy (beneficial when there are free CPU resources) with an interruption system using communications over InfiniBand. In [37] the authors present an analysis of the methodology for benchmarking nonblocking collective operations using overlapping in the latency measures, which is estimated using both time and workload measures. More recently, the successful approach of overlapping communication with costly computation has also been applied to I/O operations as shown in [38].

The above works on nonblocking collectives mainly focus on RDMA networks like InfiniBand, whereas in this paper we aim to provide an analysis of the implications of using nonblocking message passing collectives in shared memory environments.

3. NONBLOCKING COLLECTIVE LIBRARY FOR MESSAGE PASSING IN JAVA

The benefits of nonblocking collectives have been extensively studied for distributed memory systems and communication across the network. However, although shared memory architectures are becoming increasingly supported by message passing libraries, there is no previous assessment

Prepared using cpeauth.cls
DOI: 10.1002/cpe

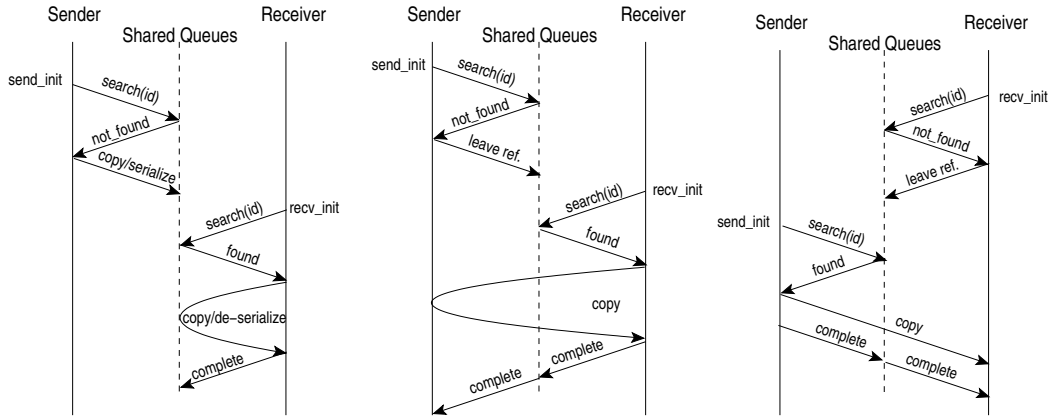
of the capabilities of the shared memory communication support to take advantage of nonblocking collectives. With this purpose, a Java message passing nonblocking collective library has been developed and integrated in the FastMPJ project [3, 4], and then it has been benchmarked in a Sandy-Bridge-based shared memory scenario. In this section, before describing the nonblocking library itself, we will address how point-to-point shared memory communications progress in FastMPJ since it is key for the implementation of the specific algorithms for nonblocking collectives on multi-core systems, introducing as well the blocking collective library currently included in FastMPJ.

3.1. Shared Memory Message Passing in FastMPJ

Shared memory communications in FastMPJ rely on intra-JVM (Java Virtual Machine) transfers among threads implemented in the `smdev` communication device [6]. This device is based on multi-threading, allowing the scheduling of a single JVM, thus saving memory and reducing overheads such as those imposed by the Garbage Collector. However, threads have to simulate the behavior of processes in order to maintain the multi-threading mechanism transparent to the user. This requires the management of the Java Class Loader hierarchy to provide each thread with a custom class loader, thus creating a namespace for each one in which each loaded class is different from the same class loaded by another thread.

Communications in `smdev` are carried out by a shared class that contains shared queues, supporting both blocking and nonblocking point-to-point primitives. Regarding nonblocking primitives, each thread runs independently, and any of the two threads involved in a communication should be able to make it progress. However, in MPJ, a message can contain either primitive types or objects and, when transferring objects, there is a serialization/de-serialization process involved. This process interferes with the class loader system and this forces to de-serialize the message within the receiver thread, which will necessarily complete a communication when serialized objects are being transferred.

A summary of the protocols used in `smdev` for nonblocking point-to-point communications is presented in Figure 1. Figure 1(a) illustrates the scenario described above about transferring objects, that is similar to the sequence for a small message of a primitive data type when the sender initiates the communication. When a small amount of data is being transferred, it is almost costless to make a copy in the shared storage ("shared queues"), so the sender can assume the communication as complete immediately. After that, the receiver will find the message and will copy it to the destination address. When sending a large message (Figure 1(b)) an extra copy will be too expensive, so a zero-copy rendez-vous protocol is used and the sender leaves a reference to the



(a) Small or serializable message, starting with Send (Eager)

(b) Large message, starting with Send (Rendez-vous)

(c) Small or large message, starting with Recv

Figure 1. Communication protocols and progression in `smdev`, the FastMPJ shared memory communication device.

DOI: 10.1002/cpe

source buffer. When the receiver starts, it makes the copy directly from this reference. However, the sender cannot assume the communication as complete until this copy has been performed. Finally, when the receiver initiates the communication (Figure 1(c)), regardless the size of the message, a zero-copy protocol is implemented. The receiver leaves a reference to its own destination buffer, and then the sender will perform the actual copy from the source to the destination address and will set the communication as complete. For a serialized message, even if the receiver initiates the communication, the sender cannot complete the transfer and the receiver is the one that de-serializes and sets it as complete (Figure 1(a)).

Finally, another remarkable feature of `smdev` is the use of a pair of queues per thread instead of global queues: one queue where a thread posts requests for pending receptions, and another one where incoming messages are posted by threads that intend to send a message to the queue owner. Having one pair of queues (pending messages and incoming messages) per thread reduces contention in queue access and eases the implementation of fine-grained instead of coarse-grained synchronization, which is the main bottleneck in other shared memory communication devices. In fact, `smdev` relies on busy-waits and small localized synchronized blocks instead of class locks that would degrade significantly the scalability on multi-core shared memory architectures.

3.2. Blocking Collective Library in FastMPJ

Collective communications have been tackled carefully since the beginning of the FastMPJ project, including a multi-core aware collective library [5] that provides several algorithms for each collective and it is able to select at runtime, via a configuration file, the most suitable algorithm depending on the number of processes and the message size. Available algorithms in FastMPJ are shown in Table I. Here blocking (*b*)/nonblocking (*nb*) refers to the use of underlying *b*/*nb* point-to-point primitives.

Table I. Collective Algorithms in FastMPJ (BT: Binomial Tree, BTe: exotic Binomial Tree, MST: Minimum Spanning Tree, FT: Flat Tree, FaT: Four-ary Tree, BDE: BiDirectional Exchange, BKT: BucKeT or Cyclic; *nb*: NonBlocking, *b*: Blocking)

Operation	Algorithms
Barrier	BT, Gather+Bcast, BTe
Bcast	MST, *nb*FT, *b*FT, FaT, Scatter(v)+Allgather(v)
Scatter(v)	MST, *nb*FT
Gather(v)	MST, *nb*FT, *b*FT
Allgather(v)	*nb*FT, *nb*BDE, *b*BKT, *nb*BKT, BTe, Gather(v)+Bcast
Alltoall(v)	*nb*FT, *b*FT
Reduce	MST, *nb*FT, *b*FT
Allreduce	*nb*FT, *b*BDE, *nb*BDE, BTe, Reduce+Bcast
Reduce-scatter	*b*BDE, *nb*BDE, *b*BKT, *nb*BKT, Reduce+Scatter(v)
Scan	*nb*FT, OneToOne

These collectives run on top of the point-to-point primitives of the communication devices, hence taking advantage of underlying point-to-point optimizations. However, `smdev` also provides its own collectives implementation that does not rely on point-to-point primitives but on shared structures from the device. Having the collective operations implemented at the communication device level enables to optimize the use of these shared structures.

The choice to make is whether to use the already existing shared queues from the device (see Section 3.1), or to create specific shared structures for the collective operations. In both cases, optimizations rely on minimizing explicit synchronizations, taking advantage of knowing in advance the communication pattern. Hence, the collective operation can avoid the use of the point-to-point protocols explained in Figure 1, implementing the one that is more suitable for the algorithm. As an example, in a Flat Tree algorithm for the broadcast operation, the root thread relies on an atomic variable to indicate the state of an ongoing execution of the collective operation, and directly inserts

DOI: 10.1002/cpe

a send request, which contains a reference to the message (thus avoiding making several copies of the message). The rest of the threads, meanwhile, are waiting on another atomic variable to be notified that they can safely receive the message. Once the notification is received, they lock their own queue to find the request, and copy the message directly from the reference left by the root. In this case, the use of busy-waits as a notification system establishes the order of operation, avoiding the need to check and lock the queues for arrived messages until the message is actually ready to be received; hence, unnecessary locks and searches on the shared queues are avoided. Moreover, although the message is small, the rendez-vous protocol of Figure 1(b) is used, since each thread will perform its own copy of the data.

The main difference among the use of the existing shared queues from the device and the use of specific shared memory structures for collective operations is that the latter avoids interfering with the shared queues of the device that are also used for point-to-point transfers, allowing further optimizations in synchronizations. This reduction of interferences will be especially relevant for nonblocking collectives, when several ongoing operations can collide. Hence, the shared memory collectives implementation in `smdev` includes specific shared memory structures for collective communications.

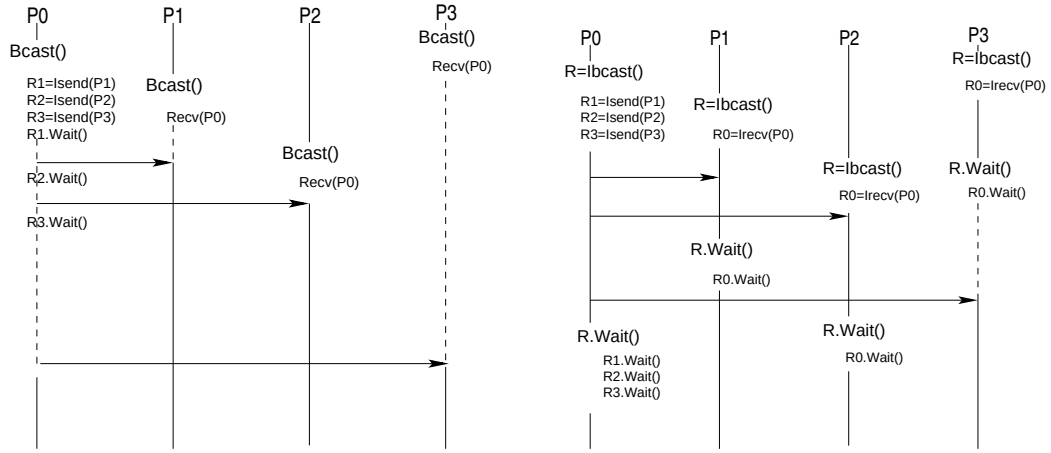
3.3. Nonblocking Collective Library for FastMPJ

Current collective operations in FastMPJ are blocking, so they do not allow the overlapping of computation and communication and, furthermore, they impose implicit synchronizations. In an environment where threads and processes are supposed to perform independent workloads, any synchronization can potentially cause major overheads. This subsection presents a high-level discussion about a generic implementation of nonblocking collectives without taking into account the underlying architecture, thus being applicable both for shared and distributed memory scenarios. Next subsection will present the specific optimizations for shared memory architectures.

An initial approach to the implementation of nonblocking collectives could be the use of a Flat Tree-based algorithm upon nonblocking point-to-point primitives, where the root is in charge of performing all the communications. Figure 2 compares this initial approach with its blocking counterpart (using also nonblocking point-to-point primitives, see *nb*FT in Table I) for a broadcast in an example scenario using four processes. Dotted lines indicate that the process has to wait and it is not able to perform any other computation while the operation is not complete, whereas continuous lines represent useful computation. Figure 2(a) represents the blocking version of the algorithm, where the `Wait` operations are immediately invoked after the point-to-point primitives, thus blocking the calling processes until the whole collective is complete. Each nonblocking point-to-point communication generates a request (Ri) over which a `Wait` operation has to be issued. In Figure 2(b), when the nonblocking collective is invoked, the point-to-point primitives are called, and the corresponding `Wait` operations can be invoked later. In this scenario, the whole nonblocking operation generates a global request (R) composed of the requests of each underlying nonblocking point-to-point primitive (Ri). Communications are therefore performed by an asynchronous progression mechanism while the process is able to continue its computation. Nevertheless, this is a naive approach with dubious benefits and hardly scalable that can collapse the progression system of the communication devices with excessive requests when the number of processes is large or when allowing concurrent nonblocking collectives.

The approach followed in our library relies on a queue of stages per process that calls a collective operation. The queues hold two types of stages: dependent and non-dependent. A dependent stage represents a communication step that has to be fulfilled before the collective operation can progress and schedule new stages. For example, in a tree-based broadcast, a process (or thread) that is not the root, can not send the message to other processes (or threads) until it has received the data from the root. A non-dependent stage, however, only requires to be complete when returning from the `Wait` method, and thus the collective operation can progress scheduling the following stages concurrently. Following the broadcast example, once a non-root process has received the message, it can send the data to the rest of its descendants concurrently, without waiting for one communication to finish before sending it to the next descendant. Hence, the use of non-dependent stages enables

DOI: 10.1002/cpe



(a) Flat Tree Blocking Bcast (*nb*FT) (b) Flat Tree Nonblocking Bcast

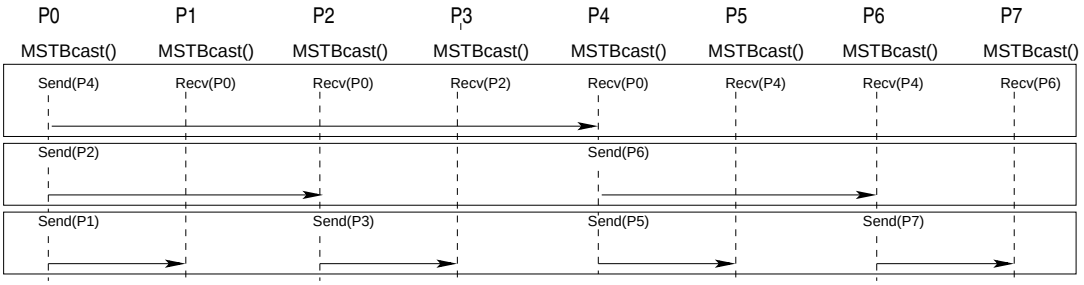
Figure 2. Flat Tree-based Bcast

the scheduling of several stages that can issue nonblocking point-to-point primitives simultaneously. With this mechanism, it is possible to implement the algorithms from Table I by splitting them in a stage manner to take advantage of their optimized performance.

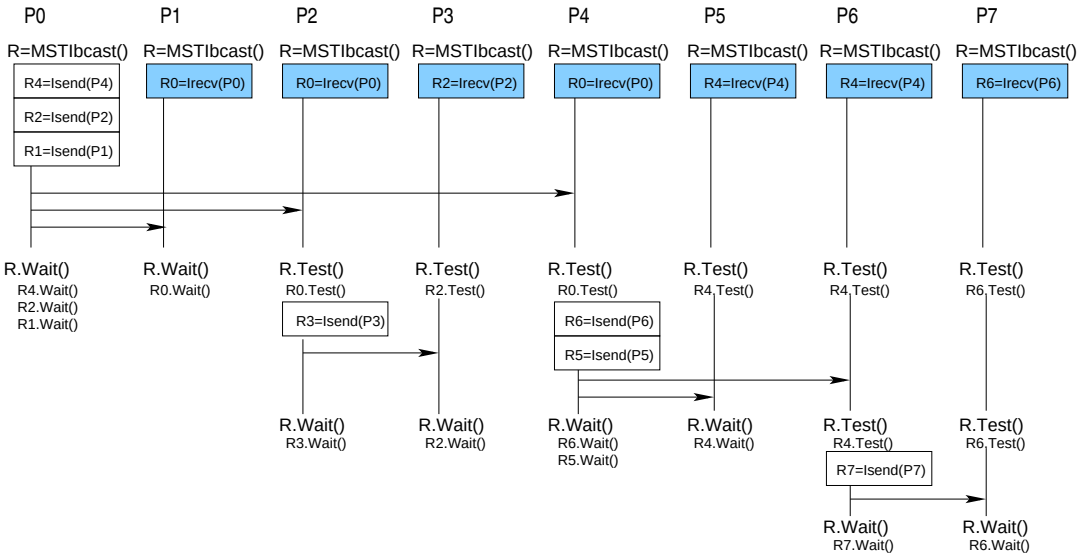
The issue that arises here is the progression of the operation. The decision to be made is if a specific mechanism is required for these operations or if it is possible to rely on each device to make progression happen, which is only possible if the operations are based on nonblocking point-to-point primitives. Since nonblocking collectives are in a very early stage of adoption, the priority is to assess the feasibility of these operations, which can be achieved relying on existing device mechanisms. Nevertheless, if the library does not create any thread to be in charge of stage progression, the user is responsible of being aware of it and making some calls to a testing function (`Test`). This function will check the stage queue and, if there is not any dependent stage pending, it will launch the subsequent stages. If there are neither pending stages nor subsequent ones, it will complete the operation. The `Wait` method is equivalent to perform several tests until the operation is finished.

Figure 3 compares a blocking Minimum Spanning Tree (MST) broadcast with a nonblocking counterpart implemented with stages for an example scenario using eight processes. The MST algorithm configures a binomial tree in which the group of processes is recursively halved. Each half has a root process that performs the communication required. The figure assumes that the collective operation is issued at the same time in every process and, in the nonblocking scenario, the calls to the `Test`/`Wait` operations are also made simultaneously. This is not a realistic scenario, but the aim of the figures is only to show the differences among both approaches. As in Figure 2, dotted lines represent idle time spent waiting for the operation to complete and continuous lines represent computation. Figure 3(a) represents the blocking implementation which uses blocking point-to-point primitives since it is a blocking recursive algorithm. This recursion causes the algorithm to be executed in three implicit steps (marked by rectangles). In Figure 3(b), the nonblocking staged implementation of the MST algorithm is represented, using dark rectangles for dependent stages and white ones for non-dependent stages. In this scenario, there are processes that schedule three or less stages, depending on how many communications they have to perform. With the purpose of ease the representation of the algorithm, we assume that a process that has already scheduled every stage calls `Wait` instead of `Test`. In addition, when `Test` is called and the stage is already finished, this stage is never tested again. It can be seen that even considering simultaneous calls, this algorithm yields less implicit synchronization and requires less ordering than the blocking one. In fact, in the blocking version every process will end almost at the same time whereas in the nonblocking one, even when the calls are simultaneous, each process can finalize the collective when its stages are complete.

DOI: 10.1002/cpe



(a) MST Blocking Bcast



(b) MST Nonblocking Bcast

Figure 3. MST-based Bcast

3.4. Optimization of the Nonblocking Collective Library for Shared Memory

The implementation based on stages is portable to every communication device in FastMPJ, but regarding shared memory the stage-based design can be combined with the specific implementation of shared memory collectives mentioned at the end of Section 3.2. The operation of the nonblocking algorithms proposed here is the same as described in Section 3.3: instead of keeping the thread waiting when the condition to progress is not yet fulfilled, the thread checks if it has to remain in the same stage or if it is able to advance to the next one. These checks are performed in the `Test` or `Wait` function.

Flat Tree algorithms have shown significant performance improvements for shared memory communications in [6], and the stage-based design can be implemented in these algorithms by using a single stage per thread in the stage queue. Moreover, like in shared memory blocking collectives, shared structures will only contain references to messages instead of real data, barely involving any memory overhead. It is thus feasible to allow the scheduling of concurrent collective operations storing references of multiple messages. This is possible through the replication on an array of the shared structure that maintains the references to data along with the semaphores that manage the stage progression. Hence, there is a limited number of concurrent operations bounded by the number of replications. This number of replications is configurable to allow the user to find a tradeoff between memory overhead and performance. The access to replicated structures is managed by a tag, a user parameter required by point-to-point operations to identify each

DOI: 10.1002/cpe

message. Collective operations based on point-to-point primitives use this parameter internally and nonblocking collectives for shared memory can take advantage of it, since it can be used as a sequence number. As the same array of shared structures is used for all collective operations, the tag parameter is a sequence number regarding all collectives.

The tag modulo the number of concurrent operations is an index in the array of structures that store the data references. If the slot of this index is free, the operation can continue and the slot will be marked as busy with the operation tag but, if not, the operation must remain in the previous stage. Nevertheless, since this array is shared by all threads, the index can be marked as busy by another thread that has started the same operation that the new thread is trying to perform. However, the use of the tag prevents this situation from causing any problem: the index is marked with the current tag, so the new thread will realize that it is occupied by the same operation and will be able to perform its stage.

To illustrate the implementation of the shared memory nonblocking collectives, Figures 4 and 5 show the pseudo-code for the main methods used in the nonblocking broadcast: the `ibcast` and `Test` calls, and the internal function to advance between stages.

Figure 4 shows the pseudo-code for the `ibcast` and `Test` operations, that can be called from the user application. Nonblocking collectives, like nonblocking point-to-point primitives, return a request over which the `Test` and `Wait` operations can be invoked. Before returning the request, this collective operation issues an advance call to make as much progression as possible. This advance call will not make the thread wait if any of the conditions prevent it from progressing, but it will just return the stage of the operation without advancing. The `Test` method also schedules the advance returning immediately even if it was not able to move forward. The `Wait` function would perform the same operation but blocking until the collective has been completed. However, it will launch an exception if the operation can not progress because of lack of resources (i.e., there is no free slot because there are more ongoing concurrent collectives than slots in the array of replicated structures). `Wait` and `Test` only cause progression of the associated collective to avoid delaying individual calls; hence, `Wait` could cause the code to deadlock if no exception is launched.

The pseudo-code of the advance method used for the broadcast is shown in Figure 5. This is the main function that controls the progression throughout stages. It uses two condition variables implemented as `AtomicInteger` type: `collectives_nbc` and `ended_collective_nbc`. The use of atomic types and operations to maintain the consistency of shared structures enables to avoid synchronizations and locks. To support a fixed number of concurrent collective operations (`NUMBER_OF_CONCURRENT_OPERATIONS`), these variables are replicated in two arrays of `AtomicInteger` indexed by the modulo of the operation tag. Hence, each operation will have an assigned slot which consists of the condition variables and a shared buffer in which the root stores the reference to the data. The `collectives_nbc` variable controls the start and end of a collective and it has three possible states: `FREE`, `INIT` and `BUSY`. In a broadcast operation, when the root finds the `collectives_nbc` variable in the `FREE` state, it sets this variable to `INIT` to mark it as occupied but not yet prepared for the rest of the threads to perform the copy. Then, after copying the reference to the message data, the root sets the condition variable to `BUSY` with the operation tag to notify that the data is ready to be copied. All threads but the root will not be able to start the communication operation until the variable is set to the operation tag by the root. The `ended_collective_nbc` variable indicates how many threads have already performed the copy and when the root would be able to reset and free the slot.

DOI: 10.1002/cpe

```
public static Request ibcast(Object buf,int root,int tag){
  int indexTag = tag % NUMBER_OF_CONCURRENT_OPERATIONS;
  int stage = ibcastAdvance(indexTag,buf,root,tag,INITIAL_STAGE);
  return new IbcastRequest(indexTag,buf,root,tag,stage);
}

public Status Test() {
  if(stage == FINAL_STAGE)
    return COMPLETE;
  else{
    stage = ibcastAdvance(indexTag, buf, root, tag, stage);
    return null;
  }
}
```

Figure 4. Pseudo-code of the `ibcast` and `Test` methods for the shared memory nonblocking broadcast

```
public static int ibcastAdvance(int indexTag,Object buf, int root,
                                 int tag, int stage){
  boolean isRoot = (getRank()==root);
  if(isRoot){
    if((stage==INITIAL_STAGE)||(stage=NO_SLOT)){
      if(!collectives_nbc[indexTag].compareAndSet(FREE,INIT){
        return NO_SLOT;
      }
      else{
        ended_collective_nbc[indexTag].set(START);
        shared_buffers[indexTag] = buf;
        collectives_nbc[indexTag].set(tag);
        stage = FIRST_ROOT_STAGE;
      }
    }
    if(stage==FIRST_ROOT_STAGE){
      if(!ended_collective_nbc[indexTag].compareAndSet(nthreads,FREE)){
        return FIRST_ROOT_STAGE;
      }
      else{
        shared_buffers[indexTag] = null;
        ended_collective_nbc[indexTag].set(FREE);
        collectives_nbc[indexTag].set(FREE);
        return FINAL_STAGE;
      }
    }
    if(stage==FINAL_STAGE) return FINAL_STAGE;
  }
  else{
    if(stage==INITIAL_STAGE){
      if(!collectives_nbc[indexTag].compareAndSet(tag,tag){
        return INITIAL_STAGE;
      }
      else{
        copy(shared_buffers[indexTag],buf);
        ended_collective_nbc[indexTag].increment();
        return FINAL_STAGE;
      }
    }
    if(stage==FINAL_STAGE) return FINAL_STAGE;
  }
}
```

Figure 5. Pseudo-code of the `ibcastAdvance` method for the shared memory nonblocking broadcast

DOI: 10.1002/cpe

4. PERFORMANCE EVALUATION

The performance evaluation of the shared memory nonblocking collectives has been carried out on a representative 16-core shared memory testbed next described. The benchmarking consists of a micro-benchmark which compares the blocking and nonblocking versions of two collective operations (broadcast and scatter), and a production application which combines I/O operations with nonblocking collectives.

The goal of this evaluation is to analyze the actual benefits of the overlapping provided by nonblocking collectives in a shared memory environment, both at the microbenchmark and application levels.

4.1. Experimental Configuration

The nonblocking collective library has been evaluated on an Intel-based shared memory system based on the Sandy-Bridge-E architecture with 2 Intel Xeon E5-2670 octa-core processors at 2.6 GHz (a total of 16 cores in the system, 32 with HyperThreading) and 64 GBytes of RAM. Each core has a 32-KByte L1 and a 256-KByte L2 cache and the eight cores in each processor share a 20-MByte Intel Smart L3 Cache. Although the system had the HyperThreading enabled, the results are shown for 16 threads since the use of the 32 available threads does not provide any benefit in terms of performance. The OS is Linux CentOs with kernel v2.6.35, OpenJDK JVM v1.6.0_20 (IcedTea6 v1.9.8) and FastMPJ v1.0 internal release.

4.2. Micro-benchmarking of MPJ Collectives

Figures 7-11 show the performance results for broadcast (Figures 7-9) and scatter (Figures 10 and 11) of a comparative benchmark among: (1) a blocking algorithm (labeled as "block" in the figures), (2) a nonblocking algorithm without overlapping computation, i.e. with an immediate call to `Wait` after calling the collective (labeled as "nbc"), used to assess the overhead introduced by the nonblocking operation; and (3) the nonblocking algorithm overlapping the communication with a synthetic workload ("nbc+overlap"). Note that in the figures "nbc" and "nbc+overlap" are always the same for each collective in order to compare the performance of each nonblocking collective with several blocking counterparts.

The benchmarks are based on the test published for LibNBC [1] and the performance evaluation methodology has been carefully designed following the recommendations addressed in [39] to avoid bias caused by side effects of the use of the JVM. Figure 6 shows the pseudo-code of the core of each benchmark to point out the differences. The required duration for the synthetic workload in Figure 6(c) is previously calculated by estimating the time that it takes to perform one iteration of the "nbc" version (i.e. the time to perform the nonblocking operation together with its corresponding `Wait`, as shown in Figure 6(b)). Moreover, the number of calls made to the `Test` method within the workload depends on its estimated duration.

```
for(i=0; i<n_iterations; i++){
        Collective();
}
```
(a) "block"

```
for(i=0; i<n_iterations; i++){
        Icollective();
        Wait();
}
```
(b) "nbc"

```
for(i=0; i<n_iterations; i++){
        Icollective();
        synthetic_workload();
        Wait();
}
```
(c) "nbc+overlap"

Figure 6. Pseudo-code of the micro-benchmarks

DOI: 10.1002/cpe

The latency is measured for the collective, `Test` and `Wait` calls, which make up the effective communication, discarding the execution time of the synthetic workload. The reason for doing this is to allow the analysis of the effective reduction in communication time, caused by the non-synchronized behavior of threads and the avoidance of interference among them. For example, for the broadcast, descendants will not check continuously the slot for message arrival, but will advance some computation, giving time to the root for completing the sending before they check again. Note that the evaluation of the effect of computation overlap on the global cost of communication and computation will be addressed in the next subsection with an MPJ application.

For the broadcast, the comparison between the shared memory nonblocking implementation (described in Section 3.4) and three blocking implementations (see Section 3.2) is shown: the blocking broadcast for shared memory (Figure 7), the MST algorithm (Figure 8), and the Flat Tree (*nb*FT) algorithm (Figure 9). Besides the specific shared memory algorithm, on which the nonblocking implementation is based, the *nb*FT algorithm has been selected as it is the point-to-point-based counterpart of the shared memory algorithm. MST has also been included for comparison purposes. As mentioned at the end of Section 3.2, the specific shared memory implementation uses the specific structures of the shared memory communication device of FastMPJ (`smdev`) directly, like the nonblocking implementation, whereas *nb*FT and MST are based on point-to-point primitives on top of `smdev`.

Regarding Figure 7, as expected, there is almost no overhead imposed on nonblocking collectives when compared to the shared memory blocking counterpart. Moreover, it can be observed that the overlapping with a computational workload reduces the time spent in the actual communication. This is because of the lack of imposed synchronization, thus when a thread calls the `Wait` function, it is more probable that other threads have already finished and they will not have to wait to perform their own operations. The differences increase from 4 MBytes on, since messages do not fit in the L3 cache (taking into account that there is a 20-MByte L3 cache shared among 8 cores), thus taking more advantage of the overlapping.

The results of the comparison with the MST algorithm in Figure 8 show that the nbc version overcomes MST, and thus the shared memory blocking implementation also overcomes MST in the same way, according to the results of Figure 7. MST introduces extra synchronizations as it is a recursive algorithm with blocking communications in each stage. The nbc+overlap version outperforms again the blocking version (MST). Finally, Figure 9 shows that, although the *nb*FT algorithm has a higher start-up latency than the algorithms based on shared memory transfers, it

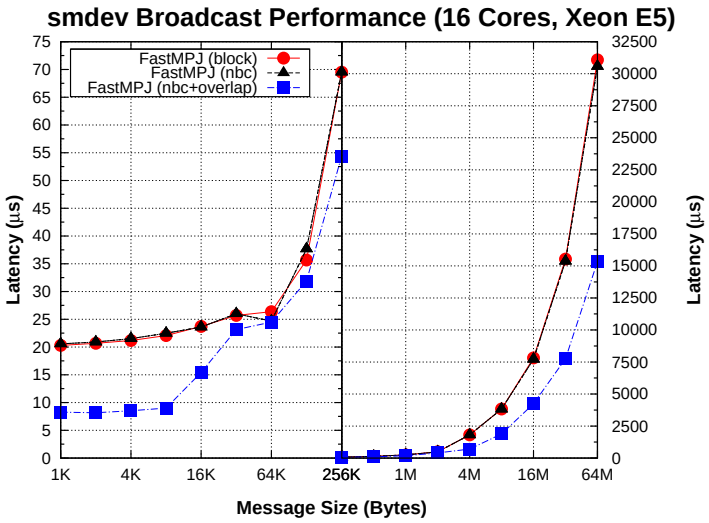


Figure 7. Shared Memory Broadcast: Blocking vs. Nonblocking

DOI: 10.1002/cpe

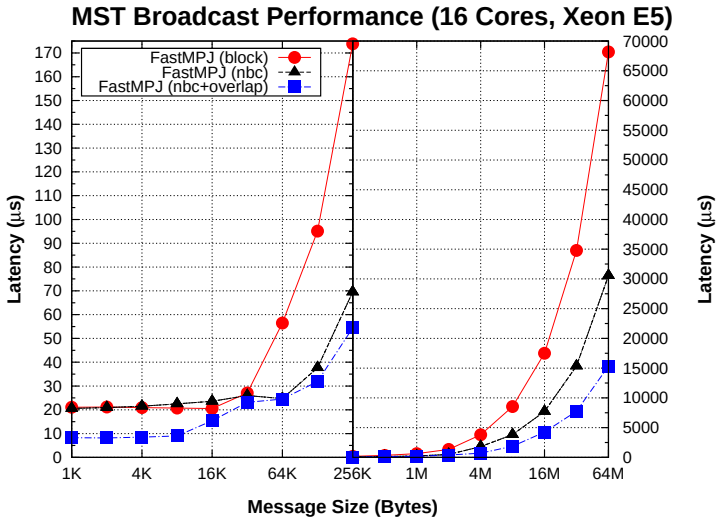


Figure 8. MST Broadcast vs. Shared Memory Nonblocking Broadcast

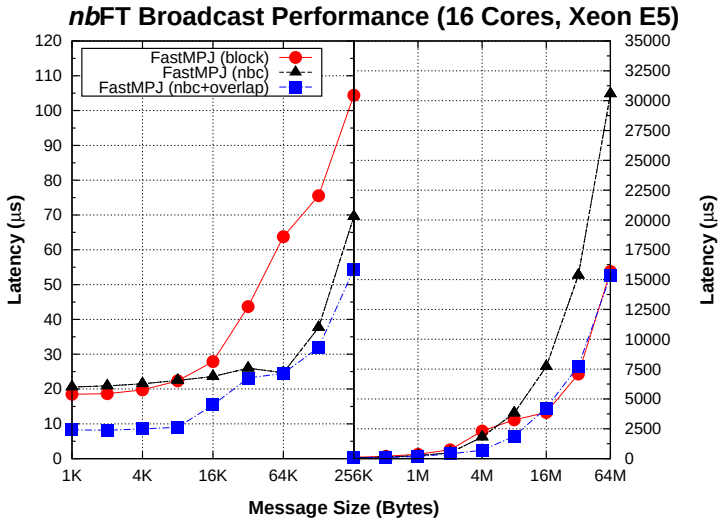


Figure 9. *nb*FT Broadcast vs. Shared Memory Nonblocking Broadcast

provides more scalability (in terms of message size) since it avoids contention in the access to the shared structures. Nevertheless, nbc+overlap achieves better performance than *nb*FT, mainly providing less start-up latency for small messages.

Regarding the scatter, the blocking versions selected were the shared memory (Figure 10) and the *nb*FT (Figure 11) algorithms. MST was discarded due to its poor performance. Results are quite similar to the ones observed for broadcast. Again, the nonblocking collective barely imposes any overhead over the shared memory blocking implementation and, when compared to the *nb*FT algorithm, it overcomes the scalability issues that contention causes on the shared memory algorithm, also reducing the latency obtained with *nb*FT, as it happened for the broadcast operation.

DOI: 10.1002/cpe

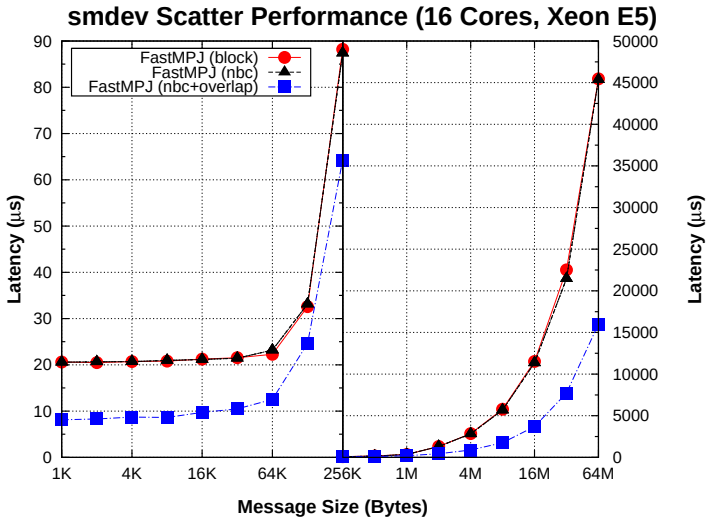


Figure 10. Shared Memory Scatter: Blocking vs. Nonblocking

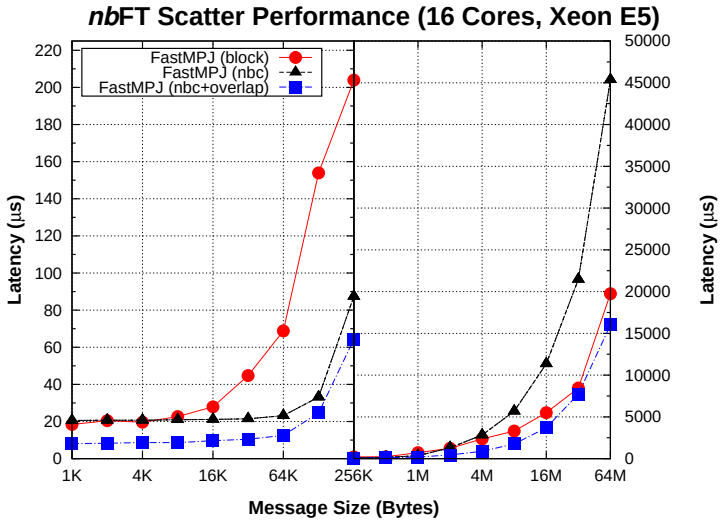


Figure 11. *nb*FT Scatter vs. Shared Memory Nonblocking Scatter

4.3. MPJ Application Performance Analysis

The application used to assess the performance of the implemented nonblocking collectives has been selected as it overlaps collective communications with computation and I/O operations. MPI includes the MPI-I/O library to deal with input/output operations and the feasibility of the use of I/O nonblocking collectives has been studied in [38]. In MPJ there are no MPJ-I/O libraries available, so parallel codes have to deal directly with the standard Java I/O libraries, generally imposing a large overhead which makes them suitable for overlapping. Although it is possible to use an extra thread to perform the I/O operations, this mechanism is far more complicated to manage than the overlapping of nonblocking collectives and I/O operations for MPJ.

The original application reads a group of zip files which represent two years of financial data from the Spanish Market of Financial Futures, including strings of information for options and futures over the IBEX-35 (Spanish exchange index), shares and the National Spanish Bond. The

DOI: 10.1002/cpe

application has to extract and read each file and process the lines. Each line is processed at the moment that it is read and available (using the `readLine` method). For the processing of each line and to evaluate the effect of the overlapping, a synthetic workload that is measured in terms of the number of operations over each line has been created. Figures 12 and 13 show the pseudo-code of the parallelization using blocking and nonblocking scatters, respectively. The parallelization with MPJ uses the Java `readChar` method (within the `readLines` method in the pseudo-codes) to read groups of chars which make up a buffer scattered among all threads. Only one thread extracts one zip file at a time and reads groups of chars until the buffer is complete. After that, it performs a scatter and each thread runs the workload over each line received (`processLines` method). Since a whole buffer of chars is scattered, the application has also to deal with the possibility that a line could be broken between two threads and it is solved by overlapping the scattered fragments. This imposes a certain overhead compared to the sequential version, but it is more efficient for the parallelization in that there is no need for serialization nor conversion of strings to arrays of chars to build the sending buffer. Finally, every thread (including the one in charge of the I/O operations) performs the operations over the received lines (`processLines`). The performance measure does not take into account the return of the results to the thread in charge of I/O operations (i.e the `MPI.Reduce` operation) because the goal is to measure the effect of using a nonblocking scatter that allows the overlapping of communications with read operations and computation.

```
InitParameters();
if(myRank==0){
        while(true){
                read=readLines(file, index, Data);
                while(read){
                        index++;
                        if(index=Send_Size){
                                MPI.Scatter(Data,myData);
                                processLines(myData);
                                index=0;
                        }
                        read=readLines(file, index, Data);
                }
                if(!openNextorClose(list_of_files,file))
                        break;
        }
        if(index<Send_Size){//Scatter needed for sending the last lines
                FillBuffer(Data);//complete the buffer up to Send_Size to
                    scatter it
                MPI.Scatter(Data);
                processLines(myData);
        }
        MPI.Scatter(FinalBuffer,myData);//notify finalization by scattering a
            special buffer
}
else{
        finished=false;
        while(!finished){
                MPI.Scatter(Data,myData);
                if(myData==FinalData)
                        finished=true;
                else
                        processLines(myData);
        }
}
MPI.Reduce(ProcessedData);
```

Figure 12. Pseudo-code of the MPJ benchmark using blocking collectives

DOI: 10.1002/cpe

Regarding the nonblocking version, the user can define the number of concurrent nonblocking scatters of the benchmark (`MaxConcurrent`) independently of the `NUMBER_OF_CONCURRENT_OPERATIONS` parameter provided by the library. However, for a correct execution, `MaxConcurrent` must be lower or equal to the library parameter. The thread in charge of I/O operations performs an `Iscatter` each time it has a buffer ready, without waiting for previous `Iscatter`s to be complete. Given that the number of concurrent `Iscatter`s is limited by the application (`MaxConcurrent` parameter), there might be no slot for a new concurrent operation. Thus, the `getNextBuffer` method will provide a free slot and, if there is none, it will iterate over existing requests making calls to `Test` in order to make pending communications progress and get a free slot. Finally, this thread will ensure that all communications are complete by calling the `test_all_requests` method, that iterates over the requests until all of them are complete.

The operation in the rest of the threads is quite different. While the thread in charge of I/O operations can keep reading and processing lines without waiting for communications to be complete, the rest of the threads need to receive the buffer before processing the lines. Since they can also trigger up to `MaxConcurrent` communication operations, progression has to be managed carefully, because, as soon as they find a communication that is complete, they have to launch the processing of the lines. These threads use three methods in charge of progression:

- `test_all_and_process`: it is called after `MaxConcurrent` iterations to check all requests once, processing the data when a complete request is found. It also checks if the buffer received is the final one.
- `getNextBuffer`: gets the next free slot. When there is none, besides calling the `Test` method (as done for the thread in charge of I/O operations), it also triggers the processing of data when if finds a complete request. As in the previous method, it checks if the buffer received is the final one.
- `test_all_and_process_final`: it is equivalent to `test_all_and_process` but, like `test_all_requests`, it iterates over the requests until all of them are complete.

Figure 14 shows the performance of the parallel application using 16 cores on the Xeon E5 testbed and different workloads. For the parallelization with blocking collectives ("block"), the Flat Tree (*nb*FT) algorithm was chosen since, although the shared memory algorithm shows better performance for small and medium size messages, *nb*FT is better for large messages, which are extensively used in this application. The nonblocking version ("nbc") allows a defined number of concurrent nonblocking scatters for both the sender and the receivers (see 4, 8 and 16 in the legend of the figure). The results are shown in terms of execution time varying the workload according to the number of operations per line. A buffer size of 512 KBytes is received by each thread, thus having an 8-MByte scattered message. The buffer size was selected to take advantage of memory locality and L3 cache size.

It can be observed that, while the overhead when there is no computation (i.e. 0 operations per line) is negligible, the use of nonblocking collectives achieves performance gains up to 30% when the number of operations (and thus the workload) increases. This is due to the overhead imposed by the implicit synchronization of the blocking collectives as opossed to the overlapping of communication and computation in the nonblocking implementation, especially when increasing the number of concurrent nonblocking collectives.

The results of this benchmark show that the use of nonblocking collectives in shared memory architectures is beneficial for communication-intensive codes that also involve large amounts of computation assigned to threads in an unbalanced way. Thus, when having a costly I/O operation and significant workloads in each thread, introducing nonblocking collectives can provide important performance benefits.

DOI: 10.1002/cpe

```
MaxConcurrent=16;
InitParameters();
InitBuffers();
if(myRank==0){
        while(true){
                read=readLines(file, index, Data);
                while(read){
                        index++;
                        if(index=Send_Size){
                            request[index_buffer]=MPI.Iscatter(Data,myData);
                            processLines(myData);
                            index=0;
                            index_buffer=getNextBuffer();//if there are no free
                                buffers, it performs tests to force operations to
                                 progress
                        }
                        read=readLines(file, index, Data);
                }
                if(!openNextorClose(list_of_files,file))
                        break;
        }
        if(index<Send_Size){//Iscatter needed for sending the last lines
                FillBuffer(index_buffer,Data);//complete the buffer up to
                    Send_Size to scatter it
                request[index_buffer]=MPI.Iscatter(Data,myData);
                processLines(myData);
        }
        index_buffer=getNextBuffer();
        request[index_buffer]=MPI.Iscatter(FinalBuffer,myData);//notify
            finalization by scattering a special buffer
        test_all_requests(request);//calls to MPI.Test until all requests are
            complete
}
else{
        finished=false;
        while(!finished){
                if(isTestTime==MaxConcurrent){
                        test_all_and_process(request,finished);//after
                            MaxConcurrent iterations, it performs tests over
                            each request, processing lines if a request is
                            complete
                        isTestTime=0;
                }
                if(finished) break;
                index_buffer=getNextBuffer();//if there are no free buffers, it
                     performs tests to force operations to progress. In this
                    case, it also processes lines if a request is complete
                if(finished) break;
                request[index_buffer]=MPI.Iscatter(Data,myData);
                isTestTime++;
        }
        finished=false;
        while(!finished){
                test_all_and_process_final(request,finished);
        }
}
MPI.Reduce(ProcessedData);
```

Figure 13. Pseudo-code of the MPJ benchmark using nonblocking collectives and concurrent Iscatters

DOI: 10.1002/cpe

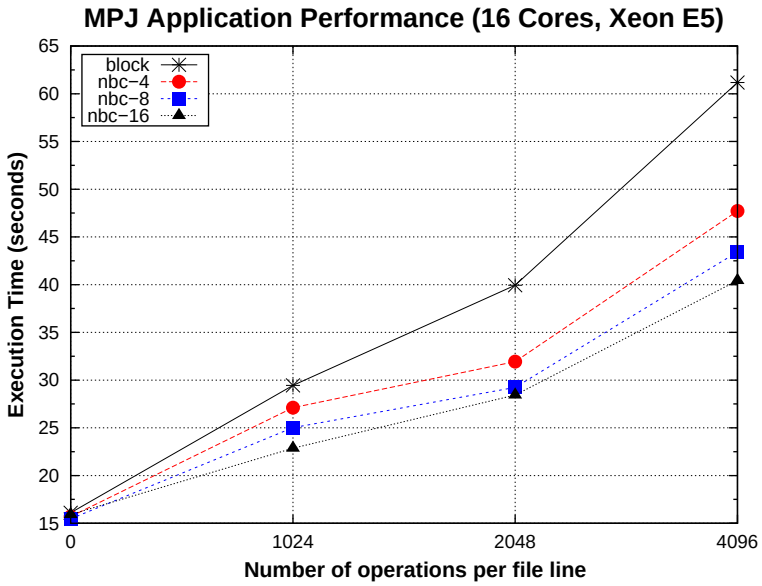


Figure 14. Performance comparison of an I/O-intensive MPJ application using Blocking ("block") and Nonblocking ("nbc") collectives

5. CONCLUSIONS

This paper has presented the development and analysis of the feasibility of the MPI 3.0 nonblocking collectives for message passing in Java focused on shared memory architectures. The performance evaluation on a representative multi-core shared memory system and the analysis of the micro-benchmarking performance results have shown that: (1) no additional overhead is imposed by using these nonblocking collectives, and (2) performance improvements are obtained when overlapping communication and computation using the nonblocking collectives in a shared memory architecture. As representative results, with the proposed nonblocking collectives there is a performance gain up to 50% for broadcast and 66% for scatter in comparison with shared memory blocking counterparts. Regarding blocking Flat Tree algorithms, which improve performance for large messages penalizing small ones, the start-up latency is reduced around 50% for both broadcast and scatter. Moreover, the nonblocking broadcast and scatter also obtain significant performance gains for large messages.

The impact on a real I/O-intensive MPJ application has been analyzed using a synthetic workload to assess the performance improvements regarding concurrent collective operations and workload overlapping. Performance results confirmed that shared memory nonblocking collectives are able to exploit the avoidance of implicit synchronization, as well as the overlapping of computation and communication. For instance, around a 30% reduction in execution time was obtained when using 16 concurrent collectives.

These results demonstrate the benefits of using nonblocking collectives in shared memory environments with multithreaded shared memory transfers, which is crucial when the trend is to increase the number of cores per processor, showing that nonblocking collectives allow communication-intensive MPJ applications to reduce significantly their overhead, thus improving the scalability of the communications.

ACKNOWLEDGEMENTS

This work has been funded by the Ministry of Science and Innovation of Spain (Project TIN2010-16735) and by the Galician Government (Projects CN2012/211 and GRC2013/055), also partially supported by FEDER funds of the European Union.

DOI: 10.1002/cpe

References

1. Hoefler T, Lumsdaine A, Rehm W. Implementation and Performance Analysis of Non-blocking Collective Operations for MPI. *Proc. 20th ACM/IEEE Intl. Conf. for High Performance Computing, Networking, Storage and Analysis (SC'07)*, Reno, NV, USA, 2007; 52 (10 pages).
2. Taboada GL, Ramos S, Expósito RR, Touriño J, Doallo R. Java in the High Performance Computing Arena: Research, Practice and Experience. *Science of Computing Programming* 2013; **78**(5):425–444.
3. Taboada GL, Touriño J, Doallo R. F-MPJ: Scalable Java Message-passing Communications on Parallel Systems. *The Journal of Supercomputing* 2012; **60**(1):117–140.
4. Expósito RR, Ramos S, Taboada GL, Touriño J, Doallo R. FastMPJ: a Scalable and Efficient Java Message-passing Library. *Cluster Computing* 2014 (In press); .
5. Taboada GL, Ramos S, Touriño J, Doallo R. Design of Efficient Java Message-passing Collectives on Multi-core Clusters. *The Journal of Supercomputing* 2011; **55**(2):126–154.
6. Ramos S, Taboada GL, Expósito RR, Touriño J, Doallo R. Design of Scalable Java Communication Middleware for Multi-core Systems. *The Computer Journal* 2013; **56**(2):214–228.
7. The Open MPI Project. Open MPI Shared Memory Communications. `http://www.open-mpi.org/faq/?category=sm` [Last visited: March 2014].
8. Ekman P, Mucci P. Design Considerations for Shared Memory MPI Implementations on Linux NUMA Systems: an MPICH/MPICH2 Case Study. *Advanced Micro Devices* July 2005; (16 pages); .
9. Buntinas D, Mercier G, Gropp W. Implementation and Evaluation of Shared-Memory Communication and Synchronization Operations in MPICH2 using the Nemesis Communication Subsystem. *Parallel Computing* 2007; **33**(9):634–644.
10. Shafi A, Carpenter B, Baker M. Nested Parallelism for Multi-core HPC Systems using Java. *Journal of Parallel and Distributed Computing* 2009; **69**(6):532–545.
11. Brightwell R, Underwood KD. An Analysis of the Impact of MPI Overlap and Independent Progress. *Proc. 18th ACM Intl. Conf. on Supercomputing (ICS'04)*, Saint Malo, France, 2004; 298–305.
12. Lawry W, Wilson C, Maccabe AB, Brightwell R. COMB: a Portable Benchmark Suite for Assessing MPI Overlap. *Proc. 14th IEEE Intl. Conf. on Cluster Computing (CLUSTER'02)*, Chicago, IL, USA, 2002; 472–475.
13. Sancho JC, Barker KJ, Kerbyson DJ, Davis K. Quantifying the Potential Benefit of Overlapping Communication and Computation in Large-scale Scientific Applications. *Proc. 20th ACM/IEEE Intl. Conf. for High Performance Computing, Networking, Storage and Analysis (SC'06)*, Tampa, FL, USA, 2006; 125 (17 pages).
14. Potluri S, Lai P, Tomko K, Sur S, Cui Y, Tatineni M, Schulz KW, Barth WL, Majumdar A, Panda DK. Quantifying Performance Benefits of Overlap using MPI-2 in a Seismic Modeling Application. *Proc. 24th ACM Intl. Conf. on Supercomputing (ICS'10)*, Tsukuba, Japan, 2010; 17–25.
15. Dongarra JJ, Otto SW, Snir M, Walker D. An Introduction to the MPI Standard. *Technical Report*, University of Tennessee, Knoxville, TN, USA 1995.
16. MPI 30 Standardization Effort. `http://meetings.mpi-forum.org/MPI_3.0_main_page.php`. [Last visited: March 2014].
17. Huang C, Lawlor O, Kalé L. Adaptive MPI. *Proc. 6th Intl. Workshop on Languages and Compilers for Parallel Computing (LCPC'03)*, College Station, TX, USA, 2003; 306–322.
18. Almási G, Heidelberger P, Archer CJ, Martorell X, Erway CC, Moreira JE, Steinmacher-Burow B, Zheng Y. Optimization of MPI Collective Communication on BlueGene/L Systems. *Proc. 19th ACM Intl. Conf. on Supercomputing (ICS'05)*, Cambridge, MA, USA, 2005; 253–262.
19. Kumar S, Dózsa G, Almási G, Heidelberger P, Chen D, Giampapa ME, Blocksome M, Faraj A, Parker J, Ratterman J, *et al.*. The Deep Computing Messaging Framework: Generalized Scalable Message Passing on the Blue Gene/P Supercomputer. *Proc. 22nd ACM Intl. Conf. on Supercomputing (ICS'08)*, Island of Kos, Greece, 2008; 94–103.
20. Kumar S, Heidelberger P, Chen D, Hines M. Optimization of Applications with Non-blocking Neighborhood Collectives via Multisends on the Blue Gene/P Supercomputer. *Proc. 24th IEEE Intl. Symposium on Parallel and Distributed Processing (IPDPS'10)*, Atlanta, GA, USA, 2010; (11 pages).
21. Kumar S, Dózsa G, Berg J, Cernohous B, Miller D, Ratterman J, Smith BE, Heidelberger P. Architecture of the Component Collective Messaging Interface. *Proc. 15th European PVM/MPI Users' Group Meeting (EuroPVM/MPI'08)*, Dublin, Ireland, 2008; 23–32.
22. Brightwell R, Goudy S, Rodrigues A, Underwood KD. Implications of Application Usage Characteristics for Collective Communication Offload. *International Journal of High Performance Computing and Networking* 2006; **4**(3-4):104–116.
23. Ferreira K, Bridges P, Brightwell R, Pedretti K. The Impact of System Design Parameters on Application Noise Sensitivity. *Proc. 12th IEEE Intl. Conf. on Cluster Computing (CLUSTER'10)*, Heraklion, Crete, Greece, 2010; 146–155.
24. Kandalla K, Subramoni H, Tomko K, Pekurovsky D, Sur S, Panda DK. High-Performance and Scalable Non-blocking All-to-All with Collective Offload on InfiniBand Clusters: a Study with Parallel 3D FFT. *Computer Science - Research and Development* 2011; **26**(3):237–246.
25. Saksena RS. On Non-blocking Collectives in 3D FFTs. *Proc. 2nd Workshop on Scalable Algorithms for Large-scale Systems (ScalA'11)*, Seattle, WA, USA, 2011; 15–18.
26. Kandalla K, Subramoni H, Vienne J, Raikar SP, Tomko K, Sur S, Panda DK. Designing Non-blocking Broadcast with Collective Offload on InfiniBand Clusters: a Case Study with HPL. *Proc. 19th IEEE Annual Symposium on High Performance Interconnects (HOTI'11)*, Santa Clara, CA, USA, 2011; 27–34.
27. Tanase G, Almási G, Archer C, Xue H. Hybrid Collective Operations on Power7 IH. *Proc. 26th ACM Intl. Conf. on Supercomputing (ICS'12)*, Venice, Italy, 2012; 215–224.
28. Nomura A, Ishikawa Y. Design of Kernel-Level Asynchronous Collective Communication. *Proc. 17th European MPI Users' Group Meeting (EuroMPI'10)*, Stuttgart, Germany, 2010; 92–101.

DOI: 10.1002/cpe

29. Nomura A, Ishikawa Y, Maruyama N, Matsuoka S. Design and Implementation of Portable and Efficient Non-blocking Collective Communication. *Proc. 12th IEEE/ACM Intl. Symposium on Cluster, Cloud and Grid Computing (CCGrid 2012)*, Ottawa, ON, Canada, 2012; (8 pages).
30. Trahay F, Brunet E, Denis A, Namyst R. A Multithreaded Communication Engine for Multicore Architectures. *Proc. 8th Workshop on Communication Architecture for Clusters (CAC'08)*, Miami, FL, USA, 2008; 167 (7 pages).
31. Trahay F, Brunet E, Denis A. An Analysis of the Impact of Multi-threading on Communication Performance. *Proc. 9th Workshop on Communication Architecture for Clusters (CAC'09)*, Rome, Italy, 2009; 142 (7 pages).
32. Schneider T, Eckelmann S, Hoefler T, Rehm W. Kernel-Based Offload of Collective Operations - Implementation, Evaluation and Lessons Learned. *Proc. 17th Intl. Europar Conf. (Europar'11)*, Bordeaux, France, 2011; 264–275.
33. Nishtala R, Zheng Y, Hargrove PH, Yelick KA. Tuning Collective Communication for Partitioned Global Address Space Programming Models. *Parallel Computing* 2011; **37**(9):576–591.
34. Hoefler T, Kambadur P, Graham RL, Shipman GM, Lumsdaine A. A Case for Standard Non-blocking Collective Operations. *Proc. 14th European PVM/MPI Users' Group Meeting (EuroPVM/MPI'07)*, Paris, France, 2007; 125–134.
35. Hoefler T, Lumsdaine A. Optimizing Non-blocking Collective Operations for InfiniBand. *Proc. 8th Workshop on Communication Architecture for Clusters (CAC'08)*, Miami, FL, USA, 2008; (8 pages).
36. Hoefler T, Lumsdaine A. Message Progression in Parallel Computing - To Thread or not to Thread? *Proc. 10th IEEE Intl. Conf. on Cluster Computing (CLUSTER'08)*, Tsukuba, Japan, 2008; 213–222.
37. Hoefler T, Schneider T, Lumsdaine A. Accurately Measuring Overhead, Communication Time and Progression of Blocking and Nonblocking Collective Operations at Massive Scale. *International Journal of Parallel, Emergent and Distributed Systems* 2010; **25**(4):241–258.
38. Venkatesan V, Chaarawi M, Gabriel E, Hoefler T. Design and Evaluation of Nonblocking Collective I/O Operations. *Proc. 18th European MPI Users' Group Meeting (EuroMPI'11)*, Santorini, Greece, 2011; 90–98.
39. Georges A, Buytaert D, Eeckhout L. Statistically Rigorous Java Performance Evaluation. *Proc. 22nd Annual ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'07)*, Montreal, QC, Canada, 2007; 57–76.

DOI: 10.1002/cpe